Truss Class Functionality

Dahlia Radif

November 1, 2018

1 Introduction

In this assignment, we are computing the beam forces for a given truss. The Truss class contains a number of functions that have allowed us to do this.

2 Methods

The following subsections explain the methodology and the corresponding Python functions used to produce the final output.

2.1 Initialisation and loading data

The `__init__` function loads the command line input. These include the *joints.dat* file, the *beams.dat* file, and an optional output file to which the truss plot will be outputted. If no such file is specified, then no plot will be generated. The data is also loaded in the two following functions, `read_joints_file` and `read_beams_file`. The inputs are the joints and beams file respectively, and `np.loadtxt` from NumPy is used to load the data, which immediately converts the file into an array of float characters.

In the `read_joints_file` function, we create a dictionary containing the joint index as key values, and a list of coordinates and zero-displacement figures as the value for each key. The main reason for this data structure is for the optional truss plot, whereby we need to access the coordinates of the two joints linked by each beam. In addition, we access the zero-displacement figure when constructing the matrix (see Section 2.4). We also create two separate lists of x and y coordinates, and two lists of the fixed forces, Fx and Fy, to be used when solving for the beam forces. In the `read_beams_file`, we simply create a dictionary whereby the key is is the beam index, and the value is a list of the two joints joined by that beam.

2.2 Truss visualisation

The function that constructs the truss plot is `PlotGeometry`, which as mentioned, is only called if the user enters a specified output file in the command line. Therefore, its input is the output file. We firstly compute the max and min of the x and y coordinates of the joints. This is to ensure that the axes used for plotting can be made large enough to accommodate the truss plot-hard-coding axes lengths may create errors in the plot visualisation. We then iterate through the beam dictionary, and for each pair of joints, we add a line to the plot corresponding to the coordinates of the two joints. The plot is formatted to appear blue, and the plot is saved to the output file. A sample truss plot from the truss1 dataset is shown below in Figure 1.

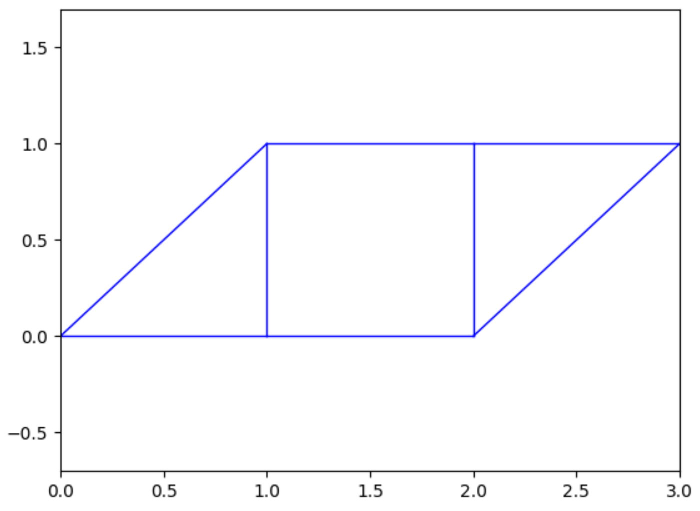

Figure 1: Truss visualisation for truss1

2.3 Computing projections along each beam

In this section of the code, the function `ComputeProjections` takes two joints and computes the Euclidean distance between their coordinates. The functions returns the normalised forces in the x and y direction, and the two joint indices (for any beam, the normalised forces in the x and y direction are the same in magnitude for the two joints connected by that beam).

2.4 Constructing the sparse matrix

Constructing the matrix in sparse form is the most technical part of this code. We discuss two functions that together, gives us the sparse matrix in COO format. Firstly, we create three empty lists: i, j and values. These contain respectively the i-index, j-index (zero based), and value corresponding to the i-j coordinate, for each matrix entry. We also create a list of joints that have zero displacement, R. The function `ConstructMatrix` calls the function `ComputeProjections` iteratively for each beam to obtain the x and y normalised forces for the two joints connected by that beam, and populates the three lists with the correct values. The number of columns of the matrix is the number of beams plus two columns for each joint with zero-displacement. The number of rows is twice the number of joints (one row for each x and y force component for each joint). For each beam, we add 4 entries in the column corresponding to that beam. For example, if beam 5 connects joints 2 and 3, then we compute the x and y components of the normalised force on those joints from beam 5, say Bx and By, and we append Bx and -Bx to the x-component rows corresponding to joint 1 and 2 respectively, and By and -By to the y-component rows corresponding to joint 1 and 2 respectively. We then iterate through the list R and append a one in the rows corresponding to joints with zero-displacement.

After we have populated the lists i, j and value, the function `CreateSparseMatrix` is called to actually create the matrix in COO format. This is done using the SciPy module `scipy.sparse.linalg` which takes as inputs the three arrays created in the previous function. To create a warning message about matrix singularity before attempting to solve the linear system, the matrix is converted to dense format and the dimensions are retrieved; if the number of columns differs from the number of rows, then the matrix is not square and hence not invertible, so a runtime error is generated.

2.5 Solving for beam forces

The function `ComputeBeamForce` creates a vector, b, which consists of the Fx followed by the Fy values created in `read_joints_file`. The vector b is populated in this order because if we were to visualise the matrix, the rows are such that joint 1 to n populate the first n rows with the x components of the normalised forces for each beam, and the next n rows are populated by y components of the normalised forces for joints 1 to n. The the vector x which contains the beam force for which we want to solve will contain the beam forces, as well as the reaction forces which we do not need to return. Before solving, we check that the matix is singular by catching a warning and returning an error message if the matrix is in fact not invertible. Otherwise, we call the function `psolve((csr_matrix(self_matrix)), b)`, which converts the matrix from COO format to CSR format, then solves for the beam forces. We then round these beam forces to three decimal places, and append them to a list (a forloop is used rather than slicing in order to be able to use the `roundResults` function.

2.6 String representation

Finally, to return the desired output in the format specified, we create a string representation of our class. This in itself uses the function, `format`, which ensures the correct alignment of positive and negative numbers in our list. For each beam, we format the beam force and add it to the string representation as required. Therefore, when printing the class from `main`, the required formatting is outputted to the user.